OVERTURE: An Object-Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments [*]

David L. Brown [†] Geoffrey S. Chesshire [†] William D. Henshaw [†]
Daniel J. Quinlan [†]

Los Alamos Unclassified Report LA-UR-97-335

Abstract

The OVERTURE Framework is an object-oriented environment for solving PDEs on serial and parallel architectures. It is a collection of C++ libraries that enables the use of finite difference and finite volume methods at a level that hides the details of the associated data structures, as well as the details of the parallel implementation. It is based on the A++/P++ array class library and is designed for solving problems on a structured grid or a collection of structured grids. In particular, it can use curvilinear grids, adaptive mesh refinement and the composite overlapping grid method to represent problems with complex moving geometry.

1 Introduction

The OVERTURE Framework is an object-oriented C++ library for solving partial differential equations (PDEs) on serial and parallel architectures. It supports finite difference and finite volume computations on a structured grid, or on a collection of structured grids. Collections of structured grids are used, for example, in the method of composite overlapping grids, with block-structured adaptive mesh refinement (AMR) algorithms, and for patched-based domain decomposition methods. This paper concentrates on the implementation of support for two of the higher-level application environments, which are the method of composite overlapping grids [7, 14] and the AMR method [1, 3, 17].

A composite overlapping grid consists of a set of logically rectangular (in 2-D) or hexahedral (in 3-D) curvlinear computational grids that overlap where they meet and together are used to describe a computational region of arbitrary complexity. This method has been used successfully over the last decade and a half, primarily to solve problems involving fluid flow in complex, often dynamically moving, geometries [3, 5, 9, 10, 21].

The data structures associated with a flexible overlapping grid solver can be quite complex. Mathematically, each component grid can be described in terms of a transformation from the unit square or cube to the coordinate space of that grid. In order to complete the description of the computational geometry, the overall composite grid also requires information specifying how the component grids communicate with each other *e.g.* through

[*]This work supported by the U.S. Department of Energy through Contract W-7405-ENG-36.

[†]Computing, Information and Communications Division, Los Alamos National Laboratory, Los Alamos, NM. Web site: `http://www.c3.lanl.gov/cic19/teams/napc/`

interpolation formulas. It is also possible for component grids to move with respect to each other as part of a time-dependent simulation. Thus, tools are required to efficiently recompute the overlap information when the grids move. In the discrete representation of such a system, for each component grid, data such as the location of the grid points, values of the transformation derivatives and volumes of the grid cells must be stored. In addition, each grid point can have attributes associated with it, such as whether it is used for discretization of the PDE or a boundary condition, if it will have values interpolated to it from another component grid, or possibly that it is not used at all. Information on where to find interpolation stencils for the interpolation points must also be stored.

The PDEs that are to be approximated can be quite complex. (A current application at Los Alamos involves low Mach-number combustion with many reacting species). The difference approximations that are used can vary from relatively simple (e.g. centered second-order finite-difference methods) to quite complex (e.g. unsplit Godunov procedures for compressible or incompressible fluid flow [5], or fully fourth-order centered finite difference methods [10]). In addition, techniques such as block structured AMR may be used to locally increase computational resolution and increase overall computational efficiency. If the simulation is to run on a parallel architecture, there are correspondingly more complexities involved in writing the code. The net result of the data structures, advanced algorithms, and modern architectures is a PDE solver code that is an extremely complex system. Successfully writing, debugging, modifying and maintaining software that implements this complex system is a daunting if not impossible task using a traditional structured programming approach and procedural languages such as Fortran or C.

An alternative to the traditional structured approach is to use object-oriented design principles and object-oriented languages like C++ to write the code [2]. With object-oriented design, the task is to develop computational "objects" that represent fundamental abstractions of elements in a computational model. Where in the structured approach, the fundamental unit of code is a subroutine or function that modifies the data in some way, in the object-oriented approach the fundamental unit is an object, described by a *class* in C++. A class contains both a description of the data structures that describe the object, as well as class member functions that operate on that data. An example of an object for a composite grid application is the composite grid itself. The class describing composite grids includes a description of the data describing the grid as well as functions that operate on that data. Examples of such functions might be those that get or put the data to a database file, add an adaptive mesh refinement grid to the data structure, or return values of parameters that describe properties of the grid.

OVERTURE is an object-oriented framework that supports applications of the type discussed above. It has been used to develop a variety of PDE solvers that use the composite overlapping grid method and support applications at Los Alamos. Among these are solvers describing incompressible, nearly incompressible and high-speed compressible fluid flow. Under development at present are solvers for internal combustion applications. The remainder of this paper discusses details of this framework.

2 Overview of the OVERTURE Classes

The main class categories that make up OVERTURE are as follows:

- **Arrays** [18]: describe multidimensional arrays using A++/P++. A++ provides the serial array objects, and P++ provides the distribution and interpretation of communication required for their data parallel execution.

- **Mappings** [13]: define transformations such as curves, surfaces, areas, and volumes. These are used to represent the geometry of the computational domain.
- **Grids** [8, 12]: define a discrete representation of a mapping or mappings. These include single grids, and collections of grids; in particular composite overlapping grids.
- **Grid functions** [12]: storage of solution values, such as density, velocity, pressure, defined at each point on the grid(s).
- **Operators** [6, 11]: provide discrete representations of differential operators and boundary conditions
- **Plotting** [16]: provides high-level plotting interface based on OpenGL.
- **Adaptive Mesh Refinement:** The AMR++ library is described in section 8.
- **Load Balancing:** The MLB load balancing library is presented in [20].

Solvers for partial differential equations are written using the above classes.

3 The A++ and P++ array classes

A++ and P++ [18] are array class libraries for performing array operations in C++ in serial and parallel environments, respectively. P++ is the principle mechanism by which the OVERTURE Framework operates in parallel, there is little code in OVERTURE outside of P++ which is specific to parallel execution. A++/P++, in a modified form to reflect collaboration with other object-oriented work at LANL, is expected to be used as the array class library within the High-Performance C++ (HPC++) effort as well [1].

A++ is a *serial* array class library similar to FORTRAN 90 in syntax, but not requiring any modification to the C++ compiler or language. A++ provides an object-oriented array abstraction specifically well suited to large scale numerical computation. It provides efficient use of multidimensional array objects which serves to both simplify the development of numerical software and provide a basis for the development of parallel array abstractions. P++ is the *parallel* array class library and shares an identical interface to A++, effectively allowing A++ serial applications to be recompiled using P++ and thus run in parallel. This provides a simple and elegant mechanism that allows serial code to be reused in the parallel environment. With the improvements in C++ compiler technology, the A++/P++ classes are presently being converted to the use of templates which provides greater flexibility, though at the likely cost of working with fewer C++ compilers.

P++ provides a data parallel implementation of the array syntax represented by the A++ array class library. To this extent it shares a lot of commonality with FORTRAN 90 array syntax and the HPF programming model. However, in contrast to HPF, P++ provides a more general mechanism for the distribution of arrays and greater control as required for the multiple grid applications represented by both the Overlapping Grid model and the Adaptive Mesh Refinement (AMR) model. Additionally, current work is addressing the addition of task parallelism as required for parallel adaptive mesh refinement.

It is unreasonable to expect that FORTRAN 90 and HPF would readily include the mechanisms required to successfully handle the requirements of adaptive mesh computations since these algorithms are still not dominant within scientific computing. Thus the

[1]see `http://www.extreme.indiana.edu/hpc++/`

use of C++'s ability to define and extend itself to application specific domains through the construction of general purpose object-oriented libraries with overloaded operators has provided a simple mechanism to explore the development of such complex parallel applications quickly and easily.

Here is a simple example code segment that solves Poisson's equation with the Jacobi method in either a serial or parallel environment using the A++/P++ classes. Notice how the Jacobi iteration for the entire array can be written in one statement.

```
  // Solve u_xx + u_yy = f by a Jacobi Iteration
Range R(0,n)                          // ... define a range of indices: 0,1,2,...,n
floatArray u(R,R), f(R,R)             // ... declare two two-dimensional arrays
f = 1.; u = 0.; h = 1./n;             // ... initialize arrays and parameters
Range I(1,n-1), J(1,n-1);             // ... define ranges for the interior

for (int iteration=0; iteration<100; iteration++)
  u(I,J) = .25*(u(I+1,J)+u(I-1,J)+u(I,J+1)+u(I,J-1)-f(I,J)*(h*h)); // ... data parallel
```

In this example, "`Range`" objects are first constructed using base, bound, and optional stride information. These are then used to build the array objects and later to specify the indexing in the final array statement. The "`floatArray`" objects are constructed with no additional information about the distribution, thus the default distribution is a "block-block" distribution of the array object's data over all the available processors. The initialization of the array objects proceeds locally and in parallel on each processor since no communication is required. The "`for`" loop is done on all processors and the only genuinely interesting parallel part of the example code is the data-parallel statement representing the Jacobi relaxation step. To be more technically accurate this statement executes using an SPMD simulation of data parallel execution. The operands in this statement are already distributed over a selected number of processors and the operations dictated by the array statement are executed in place. Communication requirements are interpreted at runtime as required to permit the dynamic redistribution of data (required for AMR applications).

In the execution of this statement, either of two modes of execution are possible depending upon the quality of the template mechanism with the C++ compiler used.

- **Expression Templates:** With the best compilers an expression template mechanism is used, work on this has provided high performance but few C++ compilers can handle the resulting template instantiations; this represents the most recent and continuing work in P++.

- **Overloaded Operators:** With other C++ compilers the right hand side operands are evaluated using the overloaded C++ binary operators. This forces each binary operator to be evaluated separately and is less efficient.

By using expression templates, a single in-lined `for` loop is generated internally and the performance is improved significantly (within 90-95% of FORTRAN 77) on cache based architectures and for certain array statements (like the Jacobi relaxation example). In contrast, using the execution of the individual binary operators, the performance is about half that of FORTRAN 77, and for cache based architectures it is worse. A++ abstracts the array operations, and P++ abstracts the details of its execution in the parallel environment, thus providing an architecture independent mechanism for the development of larger C++ libraries.

4 Mappings and Grids

The geometry of the computational domain is defined by a set of mappings, one mapping for each grid. Mappings have been designed so that an object can be easily moved by composing it with a transformation such as a translation, rotation or scaling [13]. In general, a mapping defines a transformation from R^n to R^m. In particular, mappings can define lines, curves, surfaces, volumes, rotations, coordinate stretchings , etc. The base class `Mapping` contains the data and functions that apply to all mappings. Specific types of mappings are derived from this base class. Mappings contain a variety of information and functions that can be useful for grid generators and solvers. For example, mappings contain information about their domain space, range space, boundary conditions and singularities. Mappings are easily composed, allowing coordinate stretching, rotations, translations, bodies of revolution, etc. The inverse of a mapping is always defined, either analytically or by discrete approximation.

Grids define a discrete representation of a mapping. There are several main grid classes [8, 12]. The `MappedGrid` class defines a grid for a single mapping that contains, among other things, a mapping and a mask array for cut-out regions. The `GridCollection` class defines a collection of `MappedGrid`'s. The `CompositeGrid` class defines a valid overlapping grid, which is essentially a `GridCollection` plus interpolation information. Grids contain many geometry arrays such as grid points, Jacobians , normal vectors, face areas and cell volumes.

5 Grid Functions

Grid functions represent solution values at each point on a grid or grid-collection. There is a grid function class (of `float`'s, `int`'s or `double`'s) corresponding to each type of grid [12]. So, for example, a `MappedGridFunction` lives on a `MappedGrid` and a `CompositeGridFunction` lives on a `CompositeGrid`. Grid functions are defined with up to three coordinate indices (i.e. up to three space dimensions) and up to five component indices (i.e. they can be scalars, vectors, matrices, 3-tensors,...). Since they are derived from A++ arrays, all of the array operations are defined. In the following example, a grid function is made and assigned values at all points on the grid.

```
SphereMapping sphere;                          // ... create a mapping
MappedGrid mg (sphere);                        // ... the sphere  mapping has been used to define a grid
mg.update();                                   // ... this function computes all the geometry arrays
GridFunctionParameters defaultCentering;       // ...other grid function centerings can be
                                               // ...specified through this class
// ... create a grid fn with  default centering and 2 components defined at all grid points
floatMappedGridFunction  u(mg,defaultCentering,2);
Index I1,I2,I3;
getIndex(mg.dimension,I1,I2,I3);               // ... get Index'es for all grid points

// ... set x-component to sin(x)*cos(y)
const int xComp = 0, yComp = 1;
u(I1,I2,I3,xComp) = sin(mg.vertex(I1,I2,I3,xComp))*cos(mg.vertex(I1,I2,I3,yComp));
```

Notice that when the `floatMappedGridFunction` is declared, the number of grid points does not have to be specified since this information is contained in the `MappedGrid`.

6 Operators

Operators define discrete approximations to differential operators and boundary conditions for grid functions. Many different types of approximations can be used. For example, the class MappedGridOperators [11] defines finite-difference style operators, while the class MappedGridFiniteVolumeOperators [6] defines finite-volume style operators. An operator class for incompressible flow Godunov methods has also been implemented. The Projection class computes the divergence-free part of a velocity function and is used in some of our incompressible flow codes. Here is an example using one of the operator classes:

```
...
MappedGrid mg(sphere);
MappedGridFiniteVolumeOperators op(mg);        //define operators for a MappedGrid
floatMappedGridFunction u(mg), v(mg);
u.setOperators (op);                           //associate operators with grid fn.
u = ...                                        //assign u some values
v = u.grad();                                  //compute gradient of u
v = u.laplacian();                             //compute Laplacian(u)
v = op.laplacianCoefficients();                //compute matrix for the discrete Laplacian
```

The result of the statement u.grad() is a grid function containing the gradient of u. An equivalent statement is op.grad(u). The matrix for the discrete Laplacian holds the stencil at each grid point for the Laplacian, and so is a grid function itself. This grid function can be passed to a sparse solver, for example [15].

6.1 Boundary Conditions

The programming model for boundary conditions is to use ghost points (instead of one-sided difference approximations). A library of elementary boundary conditions such as Dirichlet, Neumann, extrapolation, etc has been defined. Solvers define more complicated boundary conditions in terms of these elementary ones. The interface is quite simple, as can be seen in the following routine.

```
// ... composite grid boundary types
const int wall = 1, inflow = 3, outflow = 5, slip = 5;
void applyVelocityBoundaryConditions (floatCompositeGridFunction & v)
{
  Index allVelocityComponents;
  allVelocityComponents = Range (0,1);
  float ZERO = 0., INFLOW_VELOCITY = 1.0;  int uComponent = 0, vComponent = 1;

                  // ... set velocity to zero on walls
  v.applyBoundaryCondition (allVelocityComponents, BCTypes::dirichlet, wall, ZERO);
                  // ... extrapolate velocities at outflow
  v.applyBoundaryCondition (allVelocityComponents, BCTypes::extrapolate, outflow);
                  // ... extrapolate corners, enforce  periodic conditions, interpolate, etc.
    ... other boundary conditions ...
  v.finishBoundaryConditions();
}
```

7 An OVERTURE code for the incompressible Navier-Stokes equations

This example demonstrates the power of the OVERTURE Framework by showing a working code that solves the incompressible Navier-Stokes equations in any number of space dimensions on an overlapping grid. It is based on a cell-centered Projection method with a two-stage Runge-Kutta time integrator. A routine to initialize the velocity, initializeVelocity, and to initialize the Projection boundary conditions, setProjectionBoundaryConditions, must also be supplied to complete the code. PlotStuff [16] is the graphics package associated with OVERTURE.

```
main ()
{
  CompositeGrid cg;                             //...create composite grid
  getFromADataBase (cg, "grid.hdf");            //...read in from database (HDF) file
  cg.update ();
  Interpolant interp (cg);                      // ... initialize interpolant
  PlotStuff ps (TRUE);                          // ... initialize plotting
  ps.plot (cg);                                 // ... plot the grid
  int numberOfVelocityComponents = 2;           // ... velocities stored in q,qMid
  GridFunctionParameters cellCentered = GridFunctionParameters::cellCentered;
  floatCompositeGridFunction q    (cg, cellCentered, numberOfVelocityComponents);
  floatCompositeGridFunction qMid (cg, cellCentered, numberOfVelocityComponents);
  initializeVelocity (vortexInBox, q, cg);
  CompositeGridFiniteVolumeOperators op (cg);
  q.setOperators (op);
  qMid.setOperators (op);
  Projection projection (cg);                   // ... initialize Projection operator
  setProjectionBoundaryConditions (projection);

// ... solve Incompressible Navier-Stokes equations
  float t=0., dt=.0005, viscosity=.05; int numberOfSteps=100, frequencyOfOutput = 10;
  for (int step=0; step < numberOfSteps; step++)
  {                        // ... predict velocity at midpoint using forward Euler
    qMid = q + 0.5*dt*(-q.convectiveDerivative() + viscosity*q.laplacian());
    applyVelocityBoundaryConditions (qMid);
                          // ... correct by enforcing incompressibility constraint
    qMid = projection.project (qMid);
                          // ... predict velocity at new time using midpoint rule
    q = q + dt*(-qMid.convectiveDerivative() + viscosity*qMid.laplacian() );
    applyVelocityBoundaryConditions (q);
                          // ... correct again with projection
    q = projection.project (q);
                          // ... plot every so many timesteps
    if (step % frequencyOfOutput == 0) ps.streamLines (q);
  }
}
```

8 Adaptive Mesh Refinement - AMR++

Adaptive mesh refinement is the process of permitting local grids to be added to the computational domain and thus adaptively tailoring the resolution of the computational grid. The block-structured AMR algorithm implemented in OVERTURE provides such support for both simple problems with a single underlying grid, and problems that use the composite overlapping grid method. The AMR algorithm itself uses the multiple grid functionality provided by the basic OVERTURE classes in an essential way. AMR results is greater computational efficiency but is difficult to support. AMR++ is a library within the OVERTURE Framework which builds on top of the previously mentioned components and provides support for OVERTURE applications requiring adaptive mesh refinement. AMR++ is current work being developed and supports the adaptive regridding, transfer of data between adaptive refinement levels, parent/child/sibling operations between local refinement levels, and includes parallel AMR support. AMR++ is a parallel adaptive mesh refinement library because it is uses OVERTURE classes which derive their parallel support from the A++/P++ array class library.

9 Software Availability

The OVERTURE Framework and documentation is available for public distribution at the Web site `http://www.c3.lanl.gov/cic19/teams/napc/`. A++/P++ dates back to its first version in 1990 and has been publicly distributed since 1994; the current version was released in 1996 [19]. The OVERTURE libraries have been under development since 1994, and have been available to the public since 1996 [4]. The AMR++ classes in OVERTURE are still under development and are expected to be released in fourth quarter 1997.

References

[1] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comp. Phys., 82 (1989), pp. 64–84.

[2] G. Booch, *Object-oriented analysis and design with applications*, Addison-Wesley, 2nd ed., 1994.

[3] K. D. Brislawn, D. L. Brown, G. Chesshire, and J. S. Saltzman, *Adaptive composite overlapping grids for hyperbolic conservation laws*, LANL Unclassified Report 95-257, Los Alamos National Laboratory, 1995.

[4] K. D. Brislawn, D. L. Brown, G. S. Chesshire, and D. J. Quinlan, *Overture code*, 1996. Los Alamos National Laboratory Computer Code LA-CC-96-04.

[5] D. L. Brown, *An unsplit Godunov method for systems of conservation laws on curvilinear overlapping grids*, Math. Comput. Modelling, 20 (1994), pp. 29–48.

[6] ——, *Classes for finite volume operators and projection operators*, LANL unclassified report 96-3470, Los Alamos National Laboratory, 1996.

[7] G. Chesshire and W. D. Henshaw, *Composite overlapping meshes for the solution of partial differential equations*, J. Comp. Phys., 90 (1990), pp. 1–64.

[8] G. S. Chesshire, *Overture: the grid classes*, LANL unclassified report 96-3708, Los Alamos National Laboratory, 1996.

[9] F. C. Dougherty and J. Kuan, *Transonic store separation using a three-dimensional Chimera grid scheme*, AIAA paper 89-0637, AIAA, 1989.

[10] W. D. Henshaw, *A fourth-order accurate method for the incompressible Navier-Stokes equations on overlapping grids*, J. Comp. Phys., 113 (1994), pp. 13–25.

[11] ——, *Finite difference operators and boundary conditions for Overture, user guide, version 1.00*, LANL unclassified report 96-3467, Los Alamos National Laboratory, 1996.

[12] ——, *Grid, GridFunction and Interpolant classes for Overture, AMR++ and CMPGRD, user guide, version 1.00*, LANL unclassified report 96-3464, Los Alamos National Laboratory, 1996.

[13] ——, *Mappings for Overture: A description of the mapping class and documentation for many useful mappings*, LANL unclassified report 96-3469, Los Alamos National Laboratory, 1996.

[14] ——, *Ogen: an overlapping grid generator for Overture*, LANL unclassified report 96-3466, Los Alamos National Laboratory, 1996.

[15] ——, *Oges user guide, version 1.00, a solver for steady state boundary value problems on overlapping grids*, LANL unclassified report 96-3468, Los Alamos National Laboratory, 1996.

[16] ——, *PlotStuff: a class for plotting stuff from Overture*, LANL unclassified report 96-3893, Los Alamos National Laboratory, 1996.

[17] D. Quinlan, *Adaptive Mesh Refinement for Distributed Parallel Processors*, PhD thesis, University of Colorado, Denver, June 1993.

[18] ——, *A++/P++ manual*, LANL Unclassified Report 95-3273, Los Alamos National Laboratory, 1995.

[19] ——, *A++/P++ class libraries*, 1996. Los Alamos National Laboratory Computer Code LA-CC-96-01.

[20] D. Quinlan and M. Berndt, *MLB: Multi-level load balancing*, in this proceedings, SIAM, 1997.

[21] J. L. Steger and J. A. Benek, *On the use of composite grid schemes in computational aerodynamics*, Computer Methods in Applied Mechanics and Engineering, 64 (1987), pp. 301–320.